Shruti: A Self-Tuning Hierarchical Aggregation System*

Praveen Yalagandula
HP Labs, Palo Alto, CA
praveen.yalagandula@hp.com

Mike Dahlin
The University of Texas at Austin
dahlin@cs.utexas.edu

Abstract

Current aggregation systems either have a single inbuilt aggregation mechanism or require applications to specify an aggregation policy a priori. It is hard to predict the read and write access patterns in large systems and hence applications built on such systems suffer from inefficient network usage. We present Shruti, a system that demonstrates a general approach for self-tuning the aggregation aggressiveness to the measured workload in the system, thus optimizing the overall communication costs (e.g., the number of messages exchanged on read and write operations).

1 Introduction

Generic information aggregation frameworks such as Astrolabe [22], MDS-2 [6], and SDIMS [25] can serve as an important building block for constructing large-scale distributed services such as system management [9, 23], application placement [24], data sharing and caching [18, 20], sensor monitoring and control [11, 13], grid resource monitoring [6, 8], multicast tree formation [2, 14], and naming and request routing [5]. Such frameworks allow scalable information monitoring by forming one or more aggregation trees spanning machines in the system and by using an aggregation function at the root of each subtree to summarize the information in the subtree's leaves.

Unfortunately, existing aggregation systems use a static aggregation strategy that can perform well for some workloads but poorly for others, which prevents any system from being a truly general solution. For example, *read-dominated* attributes such as *numCPUs* tracking the number of CPUs on a physical node rarely change in value, while *write-dominated* attributes such as *numProcesses* tracking the number of processes running on a machine change quite often. An aggregation approach tuned for read-dominated attributes will consume high bandwidth when applied to write-dominated attributes. Conversely, an approach tuned for write-dominated attributes will suffer from unnecessary query latency or imprecision for read-dominated attributes. Also, the read-write access patterns may differ for different sets of nodes (spatial heterogeneity) and may change over time (temporal heterogeneity) requiring different aggregation strategies at different times and at different parts of the system [1, 19].

Most aggregation systems have a single fixed aggregation mechanism. For example, in Astrolabe [22], aggregation is performed at all levels in the aggregation tree and the aggregated values at each level are propagated to all nodes in the subtree on writes. Such a strategy might incur high communication overheads when aggregating attributes that change often. To limit the communication cost, Astrolabe throttles the rate at which information is propagated around in the system, which might lead to unnecessary inconsistency in probe responses for attributes that rarely change. In most tree based systems such as Ganglia [9] and most DHT-based systems [18, 20], aggregation is performed up to the root on writes. In MDS-2 [8], no aggregation is performed on writes but the information is aggregated on reads.

SDIMS [25] is the first aggregation framework that allows applications to control the aggregation aggressiveness in the system. SDIMS provides two knobs, `UP` and `DOWN`, that applications set at the installation phase of an aggregation function. These knobs denote how far up aggregation is performed in an aggregation tree and how far down the aggregate values at a level are propagated on updates in the system. SDIMS also allows applications to perform *continuous* probes to handle spatial and temporal heterogeneity. Though SDIMS exposes such flexibility to applications, it requires applications to know the read and write access patterns a priori to choose an appropriate strategy.

In this work, we present Shruti, a self-tuning system built on SDIMS that tracks reads and writes at different levels in an aggregation tree and dynamically decides how far up aggregation is performed and how far down aggregate values are propagated on writes. Shruti self-tunes the aggregation aggressiveness aiming to optimize the overall communication costs (the sum of read and write message costs). We

* This work is supported in part by the NSF Grant SCI-0438314 and by the University of Texas at Austin's Center for Information Assurance and Security.

propose a *lease*-based mechanism in which a node grants a lease for an aggregate value to its parent or a child to denote that it will forward any changes to that aggregate value.

Our simulation results show that at any static globally uniform read-write operation ratio in the system, Shruti incurs similar cost as an optimal static `UP` and `DOWN` strategy for that ratio. Also, in contrast to SDIMS's policy of enforcing a single strategy across all attributes using same aggregation function, Shruti tunes to different strategies for different attributes and thus achieves half-an-order magnitude lower average messages per operation. Our results show that Shruti outperforms static aggregation strategies at almost all read-write ratios when there is a spatial heterogeneity in the access patterns. Finally, we also show that Shruti efficiently tunes to temporal heterogeneity in the read and write patterns and thus reduces communication costs by 50% in comparison to a static SDIMS policy.

2 Background: SDIMS

We base our system on SDIMS [25], the first aggregation framework that allows applications to control the aggregation aggressiveness in the system. In this section, we briefly describe the aggregation abstraction exposed by SDIMS and describe how it exposes flexible aggregation mechanisms to applications.

The aggregation abstraction in SDIMS is defined across a tree spanning all nodes in the system. An example aggregation tree is illustrated in Figure 1. Each physical node in the system is a leaf and each subtree represents a logical group of nodes. An internal non-leaf node is simulated by one of the physical nodes at the leaves of the subtree rooted at that node. Each physical node has *local data* stored as a set of $(attributeType, attributeName, value)$ tuples such as *(configuration, numCPUs, 16)*, *(mcast membership, session foo, yes)*, or *(file stored, foo, myIPaddress)*. The system associates an *aggregation function* f_{type} with each attribute type, and for each level-i subtree T_i in the system, the system defines an *aggregate value* $V_{i,type,name}$ for each (attributeType, attributeName) pair as follows. For a (physical) leaf node T_0 at level 0, $V_{0,type,name}$ is the locally stored value for the attribute type and name or NULL if no matching tuple exists. Then the aggregate value for a level-i subtree T_i is the aggregation function for the type, f_{type} computed across the aggregate values of each of T_i's k children. We illustrate a simple SUM() aggregation function in Figure 1.

Given this abstraction, SDIMS [25] leverages Distributed Hash Tables (DHTs) [18, 20] to construct multiple aggregation trees, and it maps different attributes to different trees. SDIMS leverages self-organizing properties of DHTs to repair the aggregation trees as reconfigurations happen in the system.

A major innovation of SDIMS is enabling flexible aggregate computation and propagation. While previous aggregation systems [9, 18, 20, 22] implement a single static strategy, SDIMS allows applications to choose an appropriate aggregation scheme by providing three functions: install, update, and probe. *Install()* installs an aggregation function for an attribute type and specifies the update strategy that the function will use, *Update()* inserts or modifies a node's local value for an attribute, and *Probe()* obtains an aggregate value for an attribute for a specified subtree level. The install interface allows applications to specify `UP` and `DOWN` parameters along with the aggregation function. The `UP` parameter denotes until how many levels aggregation is performed on update to an attribute. The `DOWN` parameter defines for how many levels downwards the aggregated value at a level is propagated. Figure 2 (from [25]) illustrates few interesting aggregation strategies for different values of `UP` and `DOWN` in SDIMS.

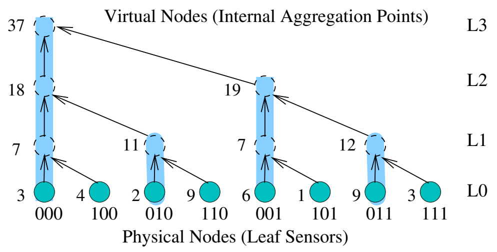

Virtual Nodes (Internal Aggregation Points)
37
18
19
7
11
7
12
3
4
2
9
6
1
9
3
000
100
010
110
001
101
011
111
L3
L2
L1
L0
Physical Nodes (Leaf Sensors)

Fig. 1. An SDIMS aggregation tree in an eight node system aggregating values using SUM() aggregation function.

Need for Self-Tuning Aggregation: Though SDIMS allows applications to explicitly control aggregation aggressiveness, an application needs to know the update and probe patterns a priori to effectively set those knobs. But, predicting the operation patterns in advance might be impossible in real distributed applications. Consider a Content Distribution Network (CDN) serving webpages for a major sporting event. This CDN can leverage SDIMS to track copies of webpages on content servers. For each copy of webpage w on a content server, the server will perform an update for attribute *(CDN, w)* with tuple $\langle IP, TimeStamp \rangle$ as value where *IP* is the IP address of the content server and *TimeStamp* is the last modified timestamp of the webpage. The aggregation function is to choose tuples with the highest timestamp value so that the global aggregate value reflects the IP addresses of servers holding the most recent copies of a webpage. When contacted by a client, a content server performs a probe for the global aggregate to locate the recent copies.

In such a system, if we choose an aggregation strategy as in typical DHT based applications, `UP`=all and `DOWN`=0, then probes for an attribute from a node need to traverse the overlay path from that node to the root of that attribute's aggregation tree. During flash-crowds, this approach can imply a drastic increase in the probe traffic. Note that simple caching might not mitigate the problem in our exam-

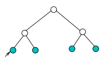
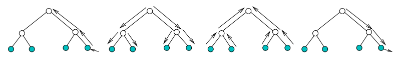
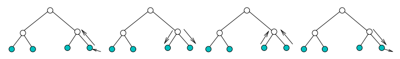
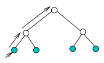
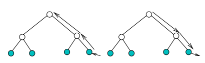
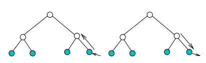
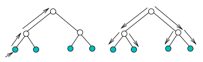
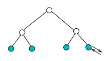
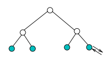
Update Strategy	On Update	On Probe for Global Aggregate Value	On Probe for Level-1 Aggregate Value
Update-Local (UP=0,DOWN=0)			
Update-Up (UP=ALL,DOWN=0)			
Update-All (UP=ALL,DOWN=ALL)			

Fig. 2. Example strategies with SDIMS `UP` and `DOWN` knobs.

ple as the content of the webpages changes frequently (as scores change in a sports event). Also note that the probe patterns can drastically vary across the individual servers of the CDN serving different portions of the Internet. So, a static `UP` and `DOWN` parameter values can be insufficient for handling spatial and temporal heterogeneity in the access patterns. Also, the update and probe rates for different attributes can vary by a large magnitude because the read popularity of webpages typically follow a Zipf-like distribution independent of the update rates [1]. But, SDIMS's aggregation abstraction imposes same `UP` and `DOWN` strategy for all attributes of the same type.

3 Shruti: Architecture

Shruti dynamically alters the propagation of aggregate values on updates so that overall communication cost—the number of messages incurred on probes and updates—is minimized. Shruti tracks updates and probes happening at all nodes in an aggregation tree and chooses an appropriate aggregation strategy based on that information. Shruti runs on each node in the aggregation tree and decides when that node will send any updates in the aggregate value to the node's parent and children. We propose a *lease*-based architecture where a node conveys its intention to keep forwarding any updates for an aggregate to another node by granting a lease for that aggregate. In the following sections, we present more details about the lease architecture, data structures that Shruti maintains to track updates and probes, how Shruti makes leasing decisions, and how Shruti handles reconfigurations.

3.1 Leases

In SDIMS, applications let all nodes in the aggregation tree know how far to propagate the aggregate values on writes by setting `UP` and `DOWN` values at the install time of an aggregation function for an attribute type. All nodes need to know this information so that when a node gets a probe it can decide whether its local state includes the aggregate value needed for answering the probe. For example, consider an attribute with `UP`=all and `DOWN`=2. We illustrate how aggregates are propagated for a part of an aggregation tree in Figure 3. Since `UP` is set to all, the level-l node A

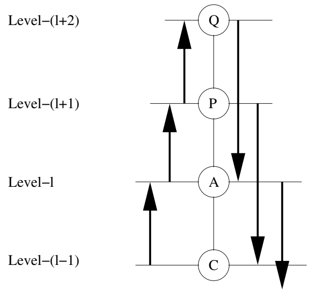


Fig. 3. Aggregate value propagation in a part of a tree for `UP`=all, `DOWN`=2.

knows that it has to propagate any updates to the level-l aggregate to its parent. Since `DOWN` is two, it propagates the level-l aggregate and also any updates it receives from its parent for the level-$(l+1)$ aggregate to its children. Similarly, this node's parent will propagate down any changes to either the level-$(l+1)$ aggregate or the level-$(l+2)$ aggregate. Now, if this node receives a probe for the level-$(l+2)$ aggregate, then it can respond without needing to further probe its parent. By knowing `UP` and `DOWN` a priori for an attribute, each node knows the rendezvous points between updates and probes for that attribute.

If we want to dynamically adapt the aggregation aggressiveness based on the workload for an attribute, then nodes in the tree need to have enough information for responding to probes. Shruti employs a *lease*-based scheme to control the level of aggregation upon updates and to let each node know how to respond to probes. A level-l node that has leases from all its children can respond to probes for the level-l aggregate without needing to probe its children. If it also has the lease from the parent for a level $l' > l$ aggregate, then it can respond to the probes for the level-l' aggregate without needing to send a probe to its parent. In Shruti, after a node A grants a level-l lease to another node B, then node A will send any changes to the level-l aggregate value until node B relinquishes that lease or node A revokes the lease.

Shruti does not exclusively work with leases to decide the propagation of aggregate values in an aggregation tree. Shruti also respects any application specified install time `UP` and `DOWN` parameters for an attribute type. Shruti uses the lease-based technique to extend the propagation of aggregate values in an aggregation tree beyond the amount of

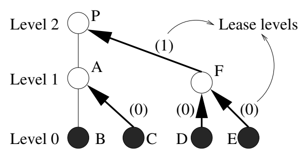


Fig. 4. An example lease state in an aggregation tree.

propagation an SDIMS static strategy allows. For example, if `UP`=all and `DOWN`=0 setting is chosen at install time of an attribute type, Shruti will always at least propagate changes in aggregate values up to the root of an aggregation tree. Shruti might further propagate aggregates down the tree to optimize communication bandwidth.

Invariants To respond correctly to probes, the leases that Shruti sets need to satisfy certain invariants. Consider the state of leases for an attribute in the Figure 4. We assume that the corresponding type is installed with `UP`=0 and `DOWN`=0. We show leases with thick arrows and denote the levels corresponding to leases in parenthesis next to the arrows. Note that in this state, node A cannot grant level-1 lease to its parent as it does not have the level-0 lease from one of its children B. Consider a situation where node A indeed granted the level-1 lease to its parent P. Since updates to the level-0 aggregate at node B are not propagated to node A, updates to the level-1 aggregate that node A propagates to its parent P will be incorrect. Any probe initiated at machines in the subtrees rooted at siblings of A (e.g., machine D) will receive an incorrect probe response. Another point to observe in Figure 4 is that node F cannot grant the level-2 lease to its children as it does not have the level-2 lease from its parent.

In the following, we present two invariants that Shruti maintains during setting up and tearing down leases to ensure correct rendezvous between probes and updates.

Invariant 1 *A level-l node can grant level-l lease to either parent or a child if and only if it has level-$(l-1)$ leases from all its children or if* `UP`$\geq l$.

Invariant 2 *A level-l node can lease level $l' > l$ to a child if and only if (i) it has the level-l' lease from its parent, (ii) if it has no parent, or (iii) if* `UP` $\geq l'$ *and* `DOWN` $\geq (l'-l)$ *holds.*

The above invariants also imply the following two conditions regarding when a node can relinquish a lease it has obtained from a neighbor of that node.

1. A node can relinquish a level-l' lease received from its parent only if it has not granted level-l' lease to any child.
2. A level-l node can relinquish a lease acquired from a child only if it has not granted a level-l lease to any other node.

3.2 Leasing Policy

Shruti keeps track of the updates and probes at each node and dynamically sets up and tears down leases between nodes to optimize overall communication costs. Briefly, it is useful for a node A to grant a level-l lease to another node B only if the number of messages that A and B exchange on probes when the lease is not granted is greater than the number of update messages that A has to forward to B after granting the lease. Note that each probe causes exchange of two messages between A and B—one for the query and one for a response; whereas each update causes only one message between two nodes. So a lease can be granted if we expect the number of probes to be even half of the number of updates. When the number of updates a node receives goes beyond two times the number of probes, then it is better to remove the lease.

We exploit dynamic adaptation of the aggregation strategies not only to optimize overall communication costs but also to trade-off bandwidth with probe response latencies. For example, if leases are set aggressively (say even when we expect probes to be far less than the half of the number of updates) but removed lazily (say remove only when the number of updates is four times more than the number of probes), then the average probe response times will be smaller but at the cost of increased communication costs. Shruti provides two knobs for the applications to control the lease aggressiveness: one knob to decide how leases are granted and another to decide how they are removed. Overall, Shruti supports and extends the flexibility provided by the static `UP` and `DOWN` SDIMS strategies.

In the following, we first describe the data structures that Shruti maintains on each node and then describe the knobs that Shruti exposes to the applications to set a leasing policy.

Data Structures Shruti maintains several logs for tracking updates and probes happening at each node. On a level-l node in an aggregation tree, Shruti maintains the following logs with timestamps:

- **LocalHistory** On each node, Shruti maintains timestamps of the most recent probe and update for each level from either its parent or any of its children. On a level-l node, the updates and probes for level $l' \geq l$ are accounted for the respective level l'. But updates received from the node's children for level-$(l-1)$ aggregate are accounted as updates for level l, since those updates affect the level-l aggregate.
- **NeighborHistory** This data structure on a node is used to track all probes and updates for all levels from each neighbor of a node (parent and children). A node maintains neighbor history for a level-l' aggregate with respect to another node B (aka history of updates and probes from node B for level l') only if node A can

Algorithm 1 OnProbe(*fromNode, reqLevel*)

```
1: timestamp ← current time
2: Add (probe, timestamp) to LocalHistory[reqLevel]
3: /* Check if invariants allow us to grant a lease for this level */
4: if canLease(reqLevel) then
5:     Add (probe, timestamp) to NeighborHistory[fromNode][reqLevel]
6:     lastUpdateTime ← timestamp of the most recent update from LocalHistory[reqLevel]
7:     numProbes ← number of probes in NeighborHistory[fromNode][reqLevel] with timestamp > lastUpdateTime
8:     if numProbes >= m then
9:         Grant lease for reqLevel aggregate to fromNode
10: else
11:     Clear all probes from NeighborHistory[fromNode][reqLevel]
```

Algorithm 2 canLease(*reqLevel*)

```
1: myLevel ← this node's level
2: if reqLevel > myLevel then
3:     if UP ≥ reqLevel AND DOWN ≥ (reqLevel−myLevel) then
4:         return true
5:     else if reqLevel ∈ LeasesReceived(parent) then
6:         return true
7:     else
8:         return false
9: else if reqLevel == myLevel then
10:     if UP ≥ myLevel then
11:         return true
12:     else
13:         for each child C ∈ children do
14:             if (reqLevel−1) ∉ LeasesReceived(C) then
15:                 return false
16:         return true
17: return false
```

Algorithm 3 OnUpdate(*fromNode, level*)

```
1: timestamp ← current time
2: Add (update, timestamp) to LocalHistory[level]
3: /* Check if invariants allow us to relinquish a lease for this level. */
4: if level ∈ LeasedFrom(fromNode) AND canRelinquish(level) then
5:     Add (update, timestamp) to NeighborHistory[fromNode][level]
6:     checkLevel ← (level < myLevel)?myLevel:level
7:     /* Note that children of a level-l node send level-(l − 1) updates and affect level-l aggregate */
8:     lastProbeTime ← timestamp of the most recent probe from LocalHistory[checkLevel]
9:     numUpdates ← number of updates in NeighborHistory[fromNode][level] with timestamp > lastProbeTime
10:     if numUpdates >= k then
11:         Relinquish lease for level aggregate from fromNode
12: else
13:     Clear all updates from NeighborHistory[fromNode][level]
```

grant the level-l' lease or has received the level-l' lease from node B.

- **LeasesGranted** and **LeasesReceived** We maintain the leases a node acquired from and leases granted to either its parent or its children in these data structures. These data structures are indexed on the neighbor.

Granting and Relinquishing leases Shruti at a node uses the history of updates and probes from another node to predict the future update-probe patterns from that node. We assume that the patterns observed in the near past reflect the near future behavior. When invariants for granting a level-l lease are satisfied at a node A, we use the following general rule to decide whether to grant the level-l lease to a node B or not—grant the lease if m probes are received from node B while no updates happened to the level-l aggregate. In Algorithm 1, we present the pseudo-code for actions performed by Shruti on receiving a probe from another node. The pseudo-code for checking whether granting a lease would violate any invariant is shown in Algorithm 2.

Algorithm 4 canRelinquish(*level*)

```
1: myLevel ← this node's level
2: if level < myLevel then
3:     level ← myLevel
4: if level ∈ LeasesGranted(parent) then
5:     return false
6: for each child C ∈ children do
7:     if level ∈ LeasesGranted(C) then
8:         return false
9: return true
```

Similar to the rule for setting a lease, we use the following rule for relinquishing a level-l' lease granted by a parent P to node A—relinquish the lease if k updates are received from the parent P while no probes are received for the level-l' aggregate. Similarly a level-$(l-1)$ lease granted by a child C is relinquished if k updates are received from the child C while no probes are received for the level-l aggregate. Note that updates for level-$(l-1)$ from a node's children affect the value of level-l aggregate at the node. The pseudo-code for Shruti's actions on receiving an update is shown in Algorithm 3. We present pseudo-code checking whether relinquishing a lease violates any invariant in Algorithm 4.

Consider an example with k=2 and m=1. A node A that can grant a level-l lease to node B grants that lease to B as soon as it gets a probe from B. And node B relinquishes that lease if it gets two updates from node A for that aggregate while not receiving any probes for that aggregate.

The two knobs k and m control how aggressively leases are set and how aggressively they are removed. A setting where k is about twice the value of m performs optimally in terms of number of messages. A large value for k and a small value for m cause leases to be set aggressively but to be removed lazily leading to better probe response times but at increased bandwidth.

3.3 Default Lease State

To be efficient and be scalable with *sparse attributes*, Shruti on each node at level-l for an attribute starts with a state where it assumes that it has level-$(l-1)$ leases from all its children and has granted a lease for the level-l aggregate to its parent.

Sparse attributes are attributes that are of interest to only few nodes in the system and only those nodes perform update or probe operations for that attribute. In most practical systems with a large number of attributes, all nodes will likely not be interested in all attributes. For example, up-

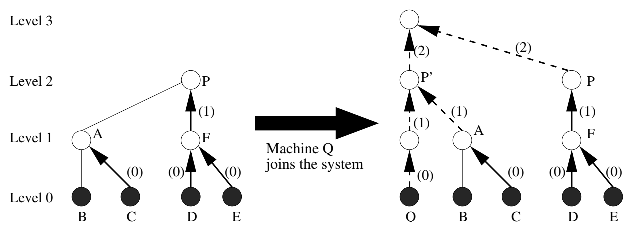


Fig. 5. Violation of Invariant 1 on reconfiguration: Node A does not have lease from B while it has lease granted to its parent. The dotted arrows represents the default leases assumed for new nodes.

dates and probes to an attribute corresponding to a multicast session with small membership will generally be done only by its members.

If nodes were to start in a state where no leases are granted for an attribute and with default UP=0 and DOWN=0, the first probe operation would incur $2.N$ messages in an N node network, as the probe would have to collect information from all nodes in the system. Thus, even for sparse attributes that are of interest to only a handful of nodes, all nodes in the system would be touched by the first few probes for that attribute until leases are set. Once leases are set, those uninterested nodes would not receive any more messages regarding the attribute. But explicitly setting leases implies that all nodes have to maintain some information about all attributes whether they are interested in that attribute or not. Hence this hypothetical initial state would undermine Shruti's ability to scale to large numbers of attributes.

Instead, by starting in a lease state in which Shruti on each node for all attributes assumes that it has leases granted from all its children and has granted a lease for the local aggregate to its parent, the unnecessary leasing information need not be explicitly maintained. Hence only nodes that are interested in an attribute and nodes helping these interested nodes in aggregation of the attribute will ever maintain any explicit information about the leases (when leases are relinquished or when leases are further granted down to children), thus achieving scalability with sparse attributes.

3.4 Reconfigurations

Reconfigurations will be a norm in any large distributed system, and in the face of reconfigurations, the invariants for leases might not continue to hold at one or more nodes in the system. Invariant 1 is violated when a level-l node in an aggregation tree acquires a new parent when it does not have level-$(l-1)$ lease from one of its children. For example, consider an aggregation tree in a four machine system shown on the left in Figure 5. In this aggregation tree, all invariants are satisfied. Suppose machine Q joins and the aggregation tree reconfigures as shown on the right in the

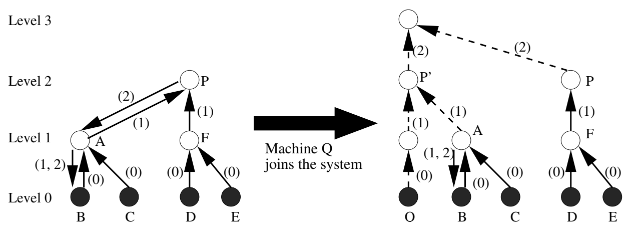


Fig. 6. Invariant 2 violation on reconfiguration. A has granted a level-2 lease to its child but does not have level-2 lease from its parent.

figure. As discussed in the previous section, new nodes start with a lease state where each level-l node assumes that it has level-$(l-1)$ lease from all its children. The dotted arrows show the default leases assumed when machine Q joins the system. At node A, Invariant 1 is violated. The parent of node A, node P', has level-1 lease from node A while node A does not have level-0 from one of its children B.

Similar to the violation of Invariant 1, Invariant 2 might be violated during reconfigurations. We show an example case where such violation occurs in Figure 6. Note that node A before reconfiguration has the level-2 lease from its parent P which it granted further down to one of its children B. When it acquired a new parent P', node A no longer has the level-2 lease from its parent but still has an outstanding level-2 lease to its child, violating Invariant 2.

In the face of reconfigurations, the goal of Shruti is to revert to a state conforming to all invariants. Observe that invariant violations occur only when a node gets a new parent. Acquiring a new child or losing an existing child does not affect the consistency of leases at a node. At a node where an invariant violation occurs, Shruti revokes leases that violate invariants. For example, when Invariant 1 is violated at a node due to acquiring a new parent, that node revokes the lease to its parent. In the example shown in Figure 5, node A revokes the level-1 lease to its new parent P'. Similarly, when Invariant 2 is violated at a node, that node revokes leases it granted to its children that violate invariants. For example, node A in Figure 6 revokes its level-2 lease to node B after it gets new parent P'.

A node that receives revocation of a lease from its parent or one of its children might have to further revoke some leases that this node granted to other nodes. For instance, when a node receives revocation of a level-l lease from its parent, it has to further revoke any level-l leases it has granted to its children. Also, when a node at level-l receives revocation of level-$(l-1)$ lease from one of its children, then it should revoke any level-l leases it has granted.

Note that while lease revocation is in progress, some of the probes might receive incorrect responses. But once the system becomes stable (no machine or network failure events or no new machine join events), Shruti on each node attains a state satisfying invariants through revoking zero

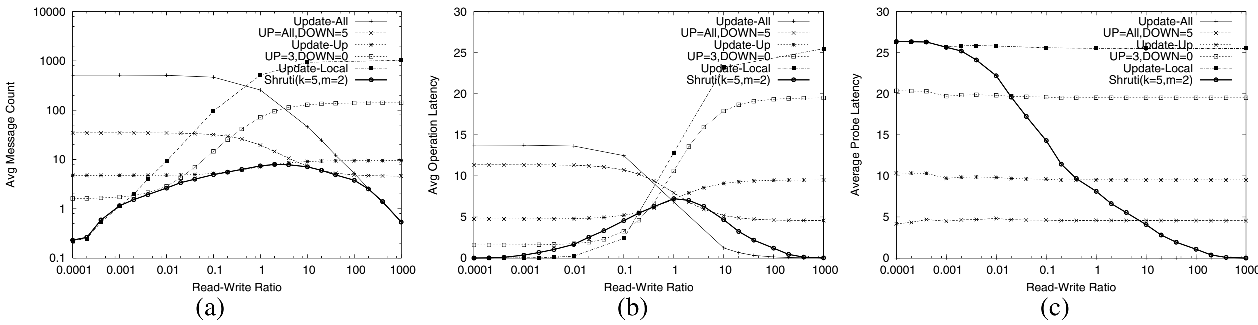


Fig. 7. Shruti vs. SDIMS static `UP` and `DOWN` for a wide range of read-to-write ratios: (a) Average message cost per operation, (b) Average latency per operation, and (c) Average probe latency.

or more leases it has granted to other nodes. Once invariants are satisfied in the lease state, all probes receive correct responses. In a separate effort [12], we address how an aggregation system can expose *network imprecision* so that a probe response can explicitly indicate whether a system is sufficiently stable for invariants such as this to hold.

4 Experimental Evaluation

We present our experimental results on Shruti comparing its performance to static up and down strategies for a wide range of update and probe patterns, referred to as read-write ratios, and with spatial and temporal heterogeneity in the access patterns. We focus on two metrics—communication cost and operation latency. Our evaluation shows that Shruti (i) adapts to the access patterns and approximates the optimal static strategy for a static and globally uniform read-write ratio, (ii) adapts to spatial heterogeneity in the access patterns across nodes to outperform any single static strategy, and (iii) quickly adapts to temporal heterogeneity such as changing access patterns.

We simulate a 512-node aggregation system, and we install a simple summation operation as the aggregation function. A write at a node for an attribute increments the previous value for the attribute at that node. We initially update each attribute at each node with a value of one. All probes are for the global aggregate value. We simulate 50000 operations for each aggregation strategy in each experiment.

Single attribute, uniform read-write ratios across nodes In this first set of experiments, we consider a single attribute and uniform read-write ratios across all nodes. For such a workload, there exists an optimal static `UP/DOWN` strategy, and we examine whether Shurti's self-tuning succeeds in balancing costs and benefits to match that strategy. In Figure 7(a), we plot the measured average message cost per operation in Shruti comparing it to different static up and down strategies in SDIMS. We use values of 5 and 2 for *k* and *m* (recall that *k* and *m* determine the aggressiveness with which leases are set and removed as explained in Section 3.2), respectively, in Shruti. Observe that at each read-write ratio, Shruti approximates the behavior of an optimal up-down SDIMS strategy at that ratio.

In Figure 7(b), we plot the average latency per operation (both updates and probes considered) in Shruti and in SDIMS. We consider each overlay hop to consume a unit of latency. Note that whereas any static strategy that behaves well at some read-to-write ratios incurs a high operation latency at other read-to-write ratios, Shruti performs well at all read-to-write ratios. For example, although the Update-all strategy performs optimally in terms of operation latency for read-to-write ratios greater than one, it incurs a high communication cost for read-write ratios less than one when compared to Shruti (Figure 7(a)).

In Figure 7(c), we plot the average probe response latencies with different read-to-write ratios for static up-down strategies and Shruti. We assume that each overlay hop has unit latency. Note that Shruti adapts to reduce overall communication bandwidth and hence incurs different latencies at different read-write ratios. When probes are rare (e.g., low read-write ratio), Shruti minimizes update cost, which increases probe latency, but as probes become more frequent, Shruti's self-tuning algorithm shifts work to the updates reducing probe latencies. All static strategies have fixed average probe response latencies.

Varying *k* and *m* in Shruti In Figure 8, we plot the average message cost observed for different values of *k* and *m* in Shruti while varying the read-to-write ratios. As expected, for large values of k and small values of m, the system adapts quickly to probes but slowly to writes; hence, it performs better at large read-write ratios but suffers at small read-write ratios. In Figure 9, we compare the average probe latency for these different strategies. Observe that the probe-favoring higher *k* compared to *m* strategies result in smaller probe latencies. We conclude two key points from these set of results: (1) *k*=5 and *m*=2 or *k*=5 and *m*=3 are good default values for *k* and *m* as Shruti performs better with these values than with any other settings and (2) applications that intend to reduce the response latency at the cost of higher bandwidth can use a more aggressive leasing

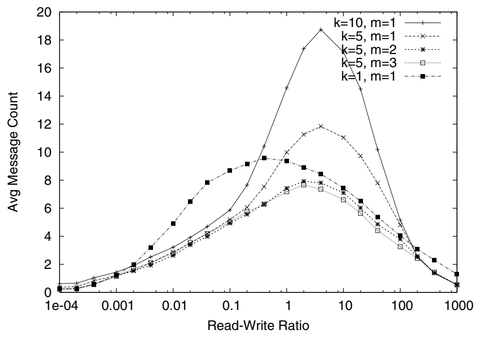


Fig. 8. Average message cost per operation in Shruti for different values of k and m.

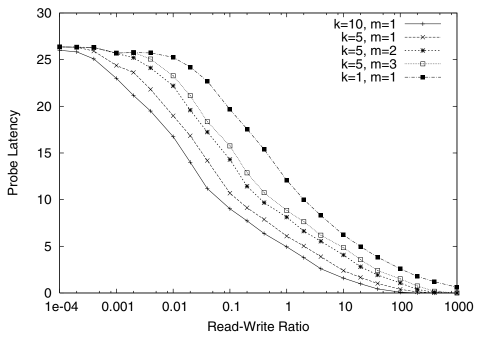


Fig. 9. Average probe latency in Shruti for different values of k and m.

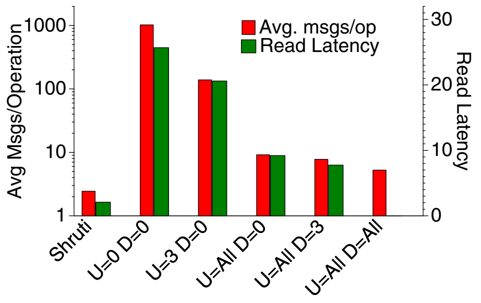


Fig. 10. Multiple attributes case: Shruti (k=5, m=2) vs SDIMS static `UP` and `DOWN` strategies.

policy by setting a high value for *k* and a small value for *m*.

Multiple attributes, Zipf-like distribution in probes Studies have shown that web accesses and P2P queries follow Zipf-like distribution [1, 19] with respect to the objects. Here, we study the performance of Shruti when probes to attributes follow Zipf-like distributions [1] (the *i*th popular attribute gets C/α^i fraction of probes) with $\alpha = 1.3$. The write operations are assigned to different attributes in a uniform way. We simulate 100 attributes, all of same type, with a global average read-to-write ratio of 100. In Figure 10, we present the average number of messages per operation and average probe latency incurred by Shruti compared to a set of SDIMS strategies with different static up and down values. Clearly, Shruti achieves both lower communication cost and smaller average probe response time than SDIMS's

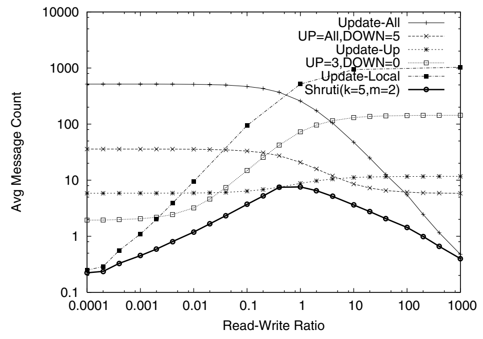


Fig. 11. Spatial heterogeneity: Average message count – Shruti vs. SDIMS strategies

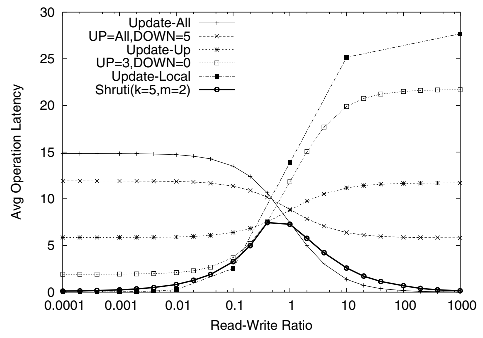


Fig. 12. Spatial heterogeneity: Average operation latency – Shruti vs. SDIMS strategies

static strategy through adapting aggregation aggressiveness separately for each individual attribute.

Spatial heterogeneity The distribution of updates and probes for an attribute will not be uniform across nodes in a real system. For example, for an attribute corresponding to a multicast session, typically only members of that multicast session perform most update and probe operations. We simulate a single attribute operation rates at nodes following a Zipf-like distribution with $\alpha = 1.3$. In Figures 11 and 12, we plot the average number of messages per operation and average operation latency incurred in Shruti in comparison to that incurred by a set of SDIMS strategies for different read-to-write ratios. Note that Shruti achieves lower communication costs compared to the first set of results in Figure 7(a) as it exploits the spatial heterogeneity to set leases such that updates and probes are propagated to only nodes interested in that attribute.

Temporal heterogeneity The read-write ratio for attributes change over time as attributes become popular and then as popularity fades. In this experiment with three phases each with 20000 operations, we start with a read-to-write ratio of 0.01 in the first phase, change it to 100 in the second phase, and revert back to 0.01 in the final phase. We measure the number of messages incurred per each operation and compare Shruti with a SDIMS Update-UP (`UP`=all and `DOWN`=0) strategy. In Figure 13, we compare the per-phase and overall average message cost for Shruti

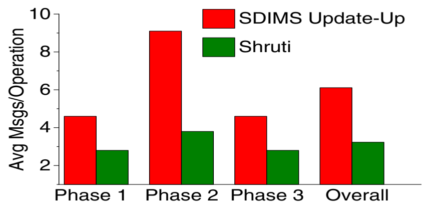


Fig. 13. Temporal heterogeneity: Average message count per operation in Shruti vs. SDIMS for three phases of the experiment.

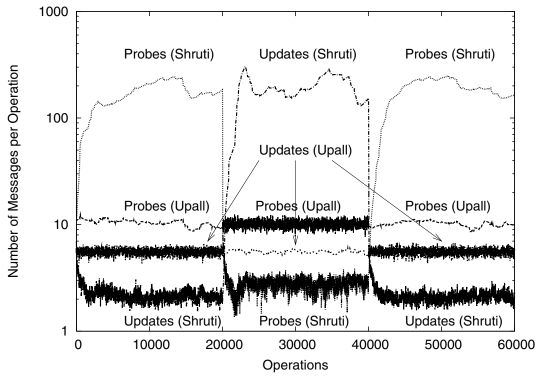


Fig. 14. Temporal heterogeneity: Number of messages per operation in Shruti vs. SDIMS `UP`=all and `DOWN`=0.

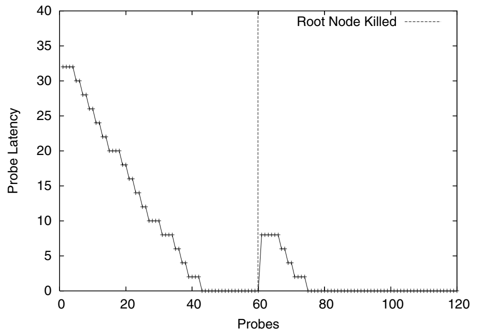


Fig. 15. Reconfiguration handling: Latency of probe operations. After sixty probes, we kill the root node of the aggregation tree.

with the SDIMS strategy. We also plot number of messages incurred per each operation in Figure 14. Note that whereas the SDIMS static scheme incurs a similar number of messages on every operation irrespective of the read-write-ratio, Shruti adapts to the observed read-write ratios and thus incurs lower communication costs.

Reconfigurations We demonstrate Shruti's reconfiguration handling capability in this experiment. As described in Section 3.4, when a virtual node in an aggregation tree loses its parent or obtains a new parent during a reconfiguration event, then the virtual node might revoke some of the leases it has granted to other nodes so that invariants are satisfied. As leases are revoked, probes might experience increased latency. With Shruti running on 1024 nodes, we first perform several write operations for an attribute at all nodes so that all leases are removed. We then perform several probes from a machine so that leases are set from all leaves to the root of the aggregation tree and on the path from the root to the probing leaf. In Figure 15, we plot the response latency of probe operations. After sixty probes, we kill the root node which causes revocation of leases on the path from the root node to the probing node. Hence the following probes suffer higher latency until the leases are set again from the new root down to the probing node.

5 Related Work

Existing aggregation systems use a static aggregation strategy that can perform well for some workloads but might perform poorly for others. Astrolabe [22] employs an *Update-All* type aggregation mechanism, DHTs and DHT based systems(e.g., [18, 20]) use an *Update-Up* mechanism, and Globus' MDS-2 [6] uses an *Update-None* mechanism.

The Controlled Update Propagation (CUP) protocol by Roussopoulos et al. [17] and Overlook [21], a name service system, are closely related to Shruti. CUP addresses a similar problem in the context of updating cached results of *get* operations in DHTs and Overlook replicates name service content along a tree to reduce lookup latency of DNS resolve queries. Though CUP, Overlook, and Shruti share the idea of using lease-based techniques, they differ in the design choices leading to different tradeoffs. First, CUP and Overlook only consider replicating the root content at other nodes; they build upon the DHT architecture and hence assume that aggregation is performed up to the root on writes. So they dynamically control the propagation of updates only downwards where as Shruti controls update propagation even towards the root. For Overlook, which is a naming service where the number of updates will be far less than the number of probes for any entry, such downward only propagation control is appropriate. In SDIMS, we consider different attributes with different behaviors. For an attribute that is updated by only a single node and no other node probes for that attribute, it is inefficient to even aggregate the data up to the root in the corresponding aggregation tree. Second, the maintenance overheads are different in these three systems. In CUP, each replicated object at a node expires unless refreshed by the parent for that object. So, the maintenance overhead is on the order of the number of objects. In Overlook and Shruti, a replicated object at a node expires if the parent that gave the lease fails. So, lease maintenance overhead involves tracking liveness of the parents of a node; hence an overhead in the order of the number of parents a node has. Whereas a node might have O(N) parents in the worst case in Overlook, each node in Shruti has to track only O($\log N$) other nodes, where N is the number of nodes in the system.

Other closely related projects are Beehive by Rama et al. [16] and SCAN by Chen et al [3]. In Beehive, no updates are considered and the goal is to place a minimum number of replicas such that all queries are satisfied with a constant communication cost, assuming queries follow a Zipf distribution. Chen et al. solve a similar problem of placing a minimum number of replicas while satisfying client QoS requirements and server constraints. Cohen et al. study replication strategies in unstructured P2P networks [4].

Lease-based techniques are employed in many distributed systems such as replicated file systems [10] and web caching and replication [7, 26]. All web replication research consider the case where updates to objects happen at a single server. Yin et al. propose volume leases [26] where a server issues a volume lease with a short timeout period and object leases with a long timeout period. This helps in reducing the communication from the server to the replicas while ensuring strong consistency guarantees. In Shruti, each node pings its neighbors frequently to check their liveness (similar to short volume lease timeout) and considers all leases that a neighbor issued to be valid as long as the neighbor is alive or the lease is either relinquished or revoked (similar to long object lease timeout).

6 Conclusions

Current aggregation systems either have single inbuilt aggregation mechanism or require applications to specify aggregation policy a priori. It is hard to predict the read and write access patterns in large systems in advance and hence such systems suffer from inefficient network usage. We present Shruti, a system that demonstrates a general approach for self-tuning the aggregation aggressiveness to the measured workload in the system optimizing the overall communication costs (e.g., the sum of costs for reads and writes). Our simulation studies demonstrate the effectiveness of dynamic adaptation in reducing communication costs for wide range of workloads. In a separate effort [15], we explore theoretical bounds for the efficacy of our tuning algorithm with leases.

References

[1] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom*, 1999.

[2] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth Multicast in a Cooperative Environment. In *SOSP*, 2003.

[3] Y. Chen, R. H. Katz, and J. D. Kubiatowicz. SCAN: a Dynamic Scalable and Efficient Content Distribution Network. In *First Intl. Conf. on Pervasive Computing*, Aug 2002.

[4] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *SIGCOMM*, 2002.

[5] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *IPTPS*, 2002.

[6] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proc HPDC*, Aug 2001.

[7] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive leases: A strong consistency mechanism for the world wide web. In *Proceedings of IEEE Infocom*, Mar. 2000.

[8] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations. In *IWQoS*, 1999.

[9] Ganglia: Distributed Monitoring and Execution System. `http://ganglia.sourceforge.net`.

[10] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proc. SOSP*, pages 202–210, 1989.

[11] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom*, 2000.

[12] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. PRISM: Precision-aware Aggregation for Scalable Monitoring. Technical Report TR-06-22, Department of Computer Sciences, UT Austin, 2006.

[13] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, 2002.

[14] A. Nandi, A. Ganjam, P. Druschel, T. E. Ng, I. Stoica, H. Zhang, and B. Bhattacharjee. SAAR: A Shared Control Plane for Overlay Multicast. In *NSDI*, 2007.

[15] C. G. Plaxton, M. Tiwari, and P. Yalagandula. Online Aggregation over Trees. In *IPDPS*, 2007.

[16] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays. In *NSDI*, March 2004.

[17] M. Roussopoulos and M. Baker. CUP: Controlled Update Propagation in Peer-to-Peer Networks. In *USENIX*, 2003.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *Middleware*, 2001.

[19] K. Sripanidkulchai. The popularity of gnutella queries and its implications on scalability, 2001. White Paper.

[20] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[21] M. Theimer and M. B. Jones. Overlook: Scalable Name Service on an Overlay Network. In *ICDCS*, 2002.

[22] R. VanRenesse, K. P. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *TOCS*, 2003.

[23] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An Information Plane for Networked Systems. In *HotNets'03*.

[24] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, Oct 1999.

[25] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM 2004*.

[26] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases to Support Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, Feb. 1999.